Annotated Inclusion Constraints for Precise Flow Analysis

Ana Milanova
Department of Computer Science
Rensselaer Polytechnic Institute
milanova@cs.rpi.edu

Barbara G. Ryder
Department of Computer Science
Rutgers University
ryder@cs.rutgers.edu

Abstract

Program flow analysis has many applications in software tools for program understanding, restructuring, verification, testing and reverse engineering. There are two important requirements for a flow analysis to be applied successfully in software tools: precision and practicality.

We propose annotated inclusion constraints—*a new general framework for formulating and implementing precise inclusion-based flow analyses. The framework can be instantiated in two dimensions: one can select a flow analysis that can be modeled using inclusion constraints (e.g., class analysis, points-to analysis) and add a dimension of precision by choosing appropriate annotations (e.g., field sensitivity, context sensitivity). The framework encompasses a large spectrum of relatively precise flow analyses.*

We formulate and implement several points-to analyses for Java as instances of the framework. The experiments show that precision dimensions such as field sensitivity and context sensitivity have significant impact on the points-to analysis and its clients. In the same time, using annotations to model these precision dimensions results in efficient and practical analysis. Therefore, flow analyses based on annotated constraints can be successfully incorporated in software tools.

1 Introduction

Program flow analysis is a technique that analyzes the source code of the program and determines properties of its run-time behavior. It is essential for building software tools for program understanding, restructuring, verification, testing and reverse engineering. There are two important requirements for a flow analysis to be successfully applied in software tools: analysis precision and analysis practicality.

Analysis precision is of crucial importance for the applicability of flow analyses in software tools because imprecise analysis results in wasted human time and decreased productivity of the developers, testers and maintainers. For example, in a static debugging tool, imprecise analysis leads to a large number of false positive warnings which leads to wasted developer time and renders the tool unusable. Another important requirement for the usability of flow analysis is *analysis practicality*. If the tool takes too long for the analysis to complete, it is likely that developers and testers will be unwilling to use it. Thus, slow analysis with large memory requirements undermines the usability of a tool and leads to decreased developer productivity. However, there is a trade-off between precision and practicality.

Because of the importance of analysis precision as well as its practicality it is necessary to investigate techniques for precise and reasonably practical flow analysis. *Inclusion constraint systems* [2, 1, 23, 6] are well studied and powerful formalism for formulating and implementing flow analyses. Analyses based on inclusion constraints examine the source code of the program and construct a system of constraints of the form $X \subseteq Y$, where X and Y are expressions representing sets; the solution of the constraint system provides information about the flow of values in the program. Analyses formulated using inclusion constraints include points-to analysis, shape analysis, closure analyses, and class analysis among others. For certain points-to analyses for C inclusion constraints have been shown to scale to large programs [6, 23, 18, 10]. However, these efficient analyses do not model dimensions of flow analysis precision such as field sensitivity (i.e., the ability of the analysis to track flow through different object fields), context sensitivity (i.e., the ability of the analysis to distinguish flow for different context of invocation of a method) and the handling of virtual dispatch. These dimensions of precision are crucial for the analysis of object-oriented programs; context and field insensitivity inherently compromise precision due to fundamental object-oriented features and programming idioms such as encapsulation, inheritance and polymorphism (Section 3.2 gives a detailed example).

Towards the goal of precise and practical flow analysis, we propose *annotated inclusion constraints*, a general framework that extends existing inclusion constraint systems. The framework is based on constraints of the form

$X \subseteq_a Y$ where the annotation a characterizes the inclusion relation. The role of the annotation is to restrict the flow in the system, i.e., there is flow from X to Z due to transitivity only if annotations a and b "match" in constraints $X \subseteq_a Y \subseteq_b Z$. The annotations model dimensions of precision in the flow analysis. The framework can be instantiated in two dimensions: one can select a flow analysis that can be modeled using inclusion constraints (e.g., class analysis, points-to analysis) and add a dimension of precision by choosing appropriate annotations (e.g., field sensitivity, context sensitivity). Thus, the framework encompasses a large spectrum of relatively precise flow analyses.

We illustrate the framework by expressing two points-to analyses for Java. Points-to analysis for Java answers the question what objects a given reference variable or a reference object field may point to. It is a fundamental flow analysis that has a wide variety of uses in software tools. The first analysis is a *field-sensitive* context-insensitive points-to analysis; it is built on top of the Java version of the field-insensitive points-to analysis for C from [2, 1, 23, 6] by using appropriate annotations that model field sensitivity and virtual dispatch. The second analysis is a novel context-sensitive points-to analysis referred to as *object-sensitive* analysis; it is built on top of the field-sensitive context-insensitive analysis by adding appropriate annotations that model flow of values for different contexts of invocation.

Previous experiments reported in [19, 13] compared field-sensitive and field-insensitive analyses; they showed that adding annotations to model field sensitivity leads to more precise and less costly analysis. In the experiments reported in this paper we compare field-sensitive context-insensitive analysis and object-sensitive analysis. The experiments show that adding annotations to model object sensitivity has substantial impact on client analyses such as *side-effect analysis* and *downcast safety analysis*. In the same time, the object-sensitive analysis remains efficient and practical. Our experiments indicate that the framework can be successfully used to implement precise and efficient flow analyses for the purposes of testing, static checking, debugging and reverse engineering.

This work has the following contributions:

- We propose annotated inclusion constraints — a new general framework that enables modeling of dimensions of precision in inclusion-based flow analysis.
- We express several flow analyses as framework instances and present experimental results that evaluate these analyses.

2 Annotated Inclusion Constraints

This section defines the general structure of the framework. Previous constraint-based flow analyses employ non-annotated inclusion constraints [6, 23]. Our new general constraint-based approach extends previous work [6, 23] by introducing constraint annotations that are used to model various dimensions of analysis precision.

Annotation Language and Operations. The set of annotations is a finite set $\mathcal{S}$. One element of the set is designated as the empty annotation and is denoted by ϵ. The following operations defined on elements of $\mathcal{S}$ are necessary for the integration of the annotations in the constraint solution procedure: $match$: $\mathcal{S} \times \mathcal{S} \rightarrow$ boolean, $concat$: $\mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$, and $transpose$: $\mathcal{S} \rightarrow \mathcal{S}$. The role of these operations in constraint resolution will be explained shortly. Operation $concat$ requires the predicate $match$ to hold. Intuitively, the set $\mathcal{S}$ and the operations acting on the elements of $\mathcal{S}$ can be instantiated to model a specific precision dimension. For example, if the annotations are used to model field sensitivity, $\mathcal{S}$ is derived from the set of field identifiers. An inclusion relation annotated with f represents flow of values through field f of a given object.

Constraint language. Flow analyses that incorporate dimensions of precision can be modeled using annotated inclusion constraints of the form $L \subseteq_a R$, where L and R correspond to some program elements and $a \in \mathcal{S}$ is chosen from the set of annotations described earlier. We use $L \subseteq R$ to denote constraints labeled with the empty annotation. L and R are set expressions, defined by the grammar:

$$L, R \rightarrow v \mid c(v_1, \ldots, v_n) \mid proj(c, i, v) \mid 0 \mid 1$$

Here v and v_i are set variables, $c(\ldots)$ are constructed terms and $proj(\ldots)$ are projection terms. Each *constructed term* is built from an n-ary constructor c. A constructor is either *covariant* or *contravariant* in each of its arguments; the role of this variance in constraint resolution will be explained shortly. Constructed terms may appear on both sides of inclusion relations. 0 and 1 represent the empty set and the universal set; they are treated as nullary constructors. *Projections* of the form $proj(c, i, v)$ are terms used to select the i-th argument of any constructed term $c(.., v_i, ..)$. Projection terms may appear only on the right-hand side of an inclusion. Intuitively, set variables, constructed terms and projection terms represent program elements and the inclusion relation represents flow of values between these elements.

Annotated Constraint Graphs. Systems of constraints can be represented as directed multi-graphs. Constraint $L \subseteq_a R$ is represented by an edge from the node for L to the node for R; the edge is labeled with the annotation a. There can be multiple edges between the same pair of nodes, each with a different annotation.

The nodes in the graph can be classified as variables, sources, and sinks. *Sources* are constructed terms that occur on the left-hand side of inclusions. *Sinks* are constructed terms or projections that occur on the right-hand side of inclusions. The graph only contains edges that represent

$c(v_1, ..., v_n) \subseteq_a c(v'_1, ..., v'_n) \Rightarrow$

$$\begin{cases} v_i \subseteq_a v'_i & \text{if } c \text{ is covariant in } i \text{ for } i\text{=1...}n \\ v'_i \subseteq_{transpose(a)} v_i & \text{if } c \text{ is contravariant in } i \text{ for } i\text{=1...}n \end{cases}$$

$c(v_1, ..., v_n) \subseteq_a proj(c, i, v) \Rightarrow$

$$\begin{cases} v_i \subseteq_a v & \text{if } c \text{ is covariant in } i \\ v \subseteq_{transpose(a)} v_i & \text{if } c \text{ is contravariant in } i \end{cases}$$

Figure 1. Rules for non-atomic constraints.

atomic constraints of the following forms: *Source* $\subseteq_a$ *Var*, *Var* $\subseteq_a$ *Var*, or *Var* $\subseteq_a$ *Sink*. If the constraint system contains a non-atomic constraint, the resolution rules from Figure 1 are used to replace this constraint with new atomic constraints, as described below.

We use annotated constraint graphs based on the *inductive form* representation [3]. Inductive form is an efficient sparse representation that does not explicitly represent the transitive closure of the constraint graph. The graphs are represented with adjacency lists $pred(n)$ and $succ(n)$ stored at each node n. Edge (n_1, n_2, a), where a is an annotation, is represented either as a predecessor edge by having $\langle n_1, a\rangle \in pred(n_2)$, or as a successor edge by having $\langle n_2, a\rangle \in succ(n_1)$, but not both. *Source* $\subseteq_a$ *Var* is always a predecessor edge and *Var* $\subseteq_a$ *Sink* is always a successor edge. *Var* $\subseteq_a$ *Var* is either a predecessor or a successor edge, based on a fixed total order τ : $Vars \rightarrow \mathcal{N}$. Edge (v_1, v_2, a) is a predecessor if and only if $\tau(v_1) < \tau(v_2)$.

Solving Systems of Annotated Inclusion Constraints. Every system of annotated inclusion constraints can be represented by an annotated constraint graph in inductive form. The system is solved by computing the closure of the graph under the following transitive closure rule:

$$\left.\begin{array}{l} \langle L, a\rangle \in pred(v) \\ \langle R, b\rangle \in succ(v) \\ match(a, b) \end{array}\right\} \Rightarrow L \subseteq_{concat(a,b)} R \quad (\text{TRANS})$$

The closure rule can be applied locally, by examining $pred(v)$ and $succ(v)$. The new transitive constraint is created *only if* the annotations of the two existing constraints "match"—that is, only if $match(a, b)$ holds, where $match$ is the specific binary predicate on the annotations. Intuitively, the annotations "color" the edges in the graph and the TRANS rule uses the colors to filter out infeasible flow. The annotation on the new constraint is produced by applying the concatenation operation to annotations a and b.

If the new constraint generated by the TRANS rule is atomic, a new edge is added to the graph. Otherwise, the resolution rules from Figure 1 are used to transform the constraint into several atomic constraints and their corresponding edges are added to the graph. For covariant arguments, the direction of flow is preserved and the annotation is also preserved. For contravariant arguments the direction of flow is reversed and a new annotation is produced by applying the specific $transpose$ operation.

The closure of a constraint graph under the TRANS rule is the *solved inductive form* of the corresponding constraint system. The least solution of the system is not explicit [3], but is easy to compute by examining all predecessors of each variable. For an annotated graph, the least solution is computed by applying transitive acyclic traversal as in [3], but the annotations are used as in rule TRANS:

$$\begin{array}{c} LS(v) = \{\langle c(\ldots), a\rangle \mid \langle c(\ldots), a\rangle \in pred(v)\} \cup \\ \{\langle c(\ldots), concat(x, y)\rangle \mid \langle u, y\rangle \in pred(v) \wedge \\ \langle c(\ldots), x\rangle \in LS(u) \wedge \ match(x, y)\} \end{array}$$

For simplicity we presented the framework using a single set of annotations; multiple sets, each modeling a different dimension of precision, can be combined by performing point-wise $match$, $concat$ and $transpose$. A detailed discussion of this issue and other relevant framework properties occurs in [13].

3 Field-sensitive Context-insensitive Points-to Analysis for Java

This section briefly outlines a popular flow- and context-insensitive points-to analysis for Java previously described in [19]. Section 3.1 discusses the semantics of the analysis and outlines its formulation in terms of annotated constraints. We add appropriate annotations to track flow through fields and to model virtual dispatch. Section 3.2 illustrates how context insensitivity inherently compromises analysis precision and stresses the need for efficient context-sensitive analyses.

3.1 Semantics and Framework Formulation

The analysis is defined in terms of three sets. Set R contains all reference variables in the analyzed program (including static variables). Set O contains names for all objects created at object allocation sites; for each allocation site s_i, there is a separate object name $o_i \in O$. Set F contains all instance fields in program classes. The analysis constructs *points-to graphs* containing two kinds of edges. Edge $(r, o_i) \in R \times O$ shows that at run time reference variable r may point to an object represented by o_i. Edge $(\langle o_i, f\rangle, o_j) \in (O \times F) \times O$ shows that field f of an object represented by o_i may point to an object represented by o_j.

For brevity, we only discuss the following kinds of statements: (1) direct assignment: $l = r$, (2) instance field write: $l.f = r$, (3) instance field read: $l = r.f$, (4) object creation: $l = new\ C$, and (5) virtual call: $l = r_0.m(r_1, ..., r_k)$. Other

$\langle l = \mathit{new}\ o_i \rangle \Rightarrow \{\mathit{ref}(o_i, v_{o_i}, \overline{v_{o_i}}) \subseteq v_l\}$
$\langle l = r \rangle \Rightarrow \{v_r \subseteq v_l\}$
$\langle l.f = r \rangle \Rightarrow \{v_l \subseteq \mathit{proj}(\mathit{ref}, 3, u),\ v_r \subseteq_f u\},\ u$ fresh
$\langle l = r.f \rangle \Rightarrow \{v_r \subseteq \mathit{proj}(\mathit{ref}, 2, u),\ u \subseteq_f v_l\},\ u$ fresh
$\langle l = r_0.m(r_1, \ldots, r_k) \rangle \Rightarrow \{v_{r_0} \subseteq_m \mathit{lam}(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)\}$

Figure 2. Field-sensitive analysis for Java.

kinds of statements (e.g. calls to static methods) are handled in a similar fashion. The analysis processes the statements and adds points-to edges to the graph. For example, at a direct assignment $l = r$ it adds edges from l to all objects that r may point to. Similarly, at an indirect write $l.f = r$ the analysis finds all objects that l may point to and adds edges labeled f from each to all objects that r may point to. At a virtual call, resolution is performed for every receiver object pointed to by r_0. The analysis uses the class of the receiver and the compile-time target of the call to determine the actual method m_j invoked at run-time. Variables $p_0, ..., p_n$ are the formal parameters of m_j; variable p_0 corresponds to the implicit parameter `this`. Variable ret_j contains the return values of m_j (we assume that each method has a unique variable that is assigned all values returned by the method; this can be achieved by auxiliary assignments).

To formulate the analysis within the framework for annotated constraints we make use of the points-to representation from [6, 23]. The set of objects pointed to by location x is represented by a set variable v_x and each object is represented by a constructed term of the form $\mathit{ref}(o, v_o, \overline{v_o})$. The last two arguments to the *ref* constructor are the same variable but with different variance. As in [6, 23] the overline bar denotes contravariant arguments. Intuitively, the second argument is used to read the fields of object o, while the third argument is used to write the fields of o. Constraint $\mathit{ref}(o, v_o, \overline{v_o}) \subseteq v_r$ represents the fact that r may point to o.

The standard constraint systems from [6, 23] allow formulation of field-insensitive points-to analysis—that is, flow through different object fields is not distinguished which leads to substantial imprecision. Our system allows us to easily add field sensitivity by using *field annotations*. The set of field annotations is the set of unique field identifiers F plus the empty field annotation ϵ_f. Operations *match* and *concat* are defined as follows:

$$\mathit{match}(f_1, f_2) = \begin{cases} \text{true} & \text{if } f_1 = \epsilon_f \text{ or } f_2 = \epsilon_f \\ \text{true} & \text{if } f_1 = f_2 \\ \text{false} & \text{otherwise} \end{cases}$$

$$\mathit{concat}(f_1, f_2) = \begin{cases} f_1 & \text{if } f_2 = \epsilon_f \\ f_2 & \text{if } f_1 = \epsilon_f \\ \epsilon_f & \text{otherwise} \end{cases}$$

We also have $\mathit{transpose}(f) = f$. Intuitively, the field annotation is propagated until it is matched with another instance of itself, after which the two instances cancel out. The analysis infers constraints of the form $\mathit{ref}(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq_f v_{o_1}$ to represent that field f of o_1 may point to object o_2.

The constraints for program statements are shown in Figure 2. The first rule expresses that object o_i, represented in the constraint system with the appropriate *ref* term, flows to the points-to set of l. Similarly, the rule for $l = r$ expresses that the point-to set of r flows to the points-to set of l. The rule for $l.f = r$ uses the first constraint to access the points-to set of l. As a result of this constraint, all $\mathit{ref}(o, v_o, \overline{v_o})$ terms to the left of v_l are matched with the projection term during resolution (i.e., the objects in the points-to set of l are found). This results in constraints of the form $u \subseteq v_o$. Combining with the second constraint generated for $l.f = r$ results in $v_r \subseteq_f u \subseteq v_o$, meaning that the points-to set of r flows into the points-to set of field f of all objects o pointed to by l. Similarly, the rule for indirect read statements uses two constraints to read the values of field f in all objects pointed to by r.

To model virtual dispatch we use *method annotations*. The rule for virtual calls in Figure 2 is based on a *lam* (lambda) constructed term which encapsulates the actual arguments and the left-hand-side variable of the call. The annotation on the constraint is a unique identifier of the *compile-time* target method of the call. To model the effects of virtual calls, we define an additional closure rule VIRTUAL. This rule encodes the semantics of virtual calls described earlier in this section and is used together with the TRANS rule to obtain the solved form of the constraint system. VIRTUAL is applied whenever we have two constraints of the form $\mathit{ref}(o, v_o, \overline{v_o}) \subseteq v,\ v \subseteq_m \mathit{lam}(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$. As described in Section 2, the edge from the *ref* term is a predecessor edge, and the edge to the *lam* term is a successor edge. Thus, the VIRTUAL closure rule can be applied locally, by examining sets $\mathit{pred}(v)$ and $\mathit{succ}(v)$. Whenever two such constraints are detected, the analysis uses a precomputed lookup table to find the lambda term for the run-time method corresponding to object o and compile-time target method m. The result of applying VIRTUAL are two new constraints:

$$\begin{array}{rcl} \mathit{ref}(o, v_o, \overline{v_o}) & \subseteq & v_{p_0} \\ \mathit{lam}(\overline{v_{p_0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_k}}, v_{ret}) & \subseteq & \mathit{lam}(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l) \end{array}$$

The first constraint creates the association between `this` of the invoked method and the receiver object. The second constraint immediately resolves to $v_{r_i} \subseteq v_{p_i}$ (for $i \geq 1$) and $v_{ret} \subseteq v_l$, plus the trivial constraint $0 \subseteq v_{p_0}$.

3.2 Imprecision of Context-insensitive Analysis

Context-insensitive analysis is imprecise for many fundamental object-oriented features and programming idioms such as encapsulation, inheritance, and containers and maps. For example, if field f is written through a method invocation, during context-insensitive analysis field f of *each* receiver will point to *all* objects passed as arguments to the

```
  class X { void n() {...} }
  class Y extends X { void n() {...} }
  class Z extends X { void n() {...} }
  class A {
     X f;
1    A(X xa) { this.f = xa; } }
  class B extends A {
2    B(X xb) { super(xb); ...  }
     ...  }
  class C extends A {
3    C(X xc) { super(xc); ...  }
     ...  }
4  s1:Y y = new Y();
5  s2:Z z = new Z();
6  s3:B b = new B(y);
7  s4:C c = new C(z);
```

Figure 3. Field write through a superclass.

method which writes that value of f. As a result, context-insensitive analysis can incur significant imprecision.

The detailed example in Figure 3 illustrates basic object-oriented features (inheritance and encapsulation) for which context-insensitive analysis produces imprecise results. At line 2, `B.this` points to o_3 and `B.xb` points to o_1. After the analysis processes the call to the superclass constructor, `A.this` and `A.xa` point to o_3 and o_1, respectively. Because of the call at line 3, `A.this` will point to o_4 and `A.xa` will point to o_2. Thus, at statement `this.f=xa` at line 1, spurious edges $(\langle o_3, f\rangle, o_2)$ and $(\langle o_4, f\rangle, o_1)$ are added to the graph. This imprecision usually propagates further and affects both the points-to analysis and its clients.

Instance field access through method invocation, initialization through a superclass constructor, and the use of containers and maps are several inherent object-oriented features and idioms for which context-insensitive analysis produces imprecise results. The use of these features and idioms is widespread. Thus, it is necessary to develop techniques for practical context-sensitive analysis.

4 Object-sensitive Analysis

Section 4.1 briefly describes the semantics of *object-sensitive analysis*, a new form of context-sensitive analysis for object-oriented languages [14, 13]. With object sensitivity, instance methods (i.e., non-static methods) and constructors are analyzed separately for objects on which such methods may be invoked. Section 4.2 shows how context sensitivity can be added to the analysis from 3.1 by using appropriate annotations that filter infeasible flow due to imprecise handling of different contexts of method invocation.

4.1 Semantics of Object-sensitive Analysis

The semantics of object-sensitive analysis is defined in terms of a set of contexts $\mathcal{C}$, a set of replicated object names O' and a set of replicated reference variables R'.

Object contexts. Let S be the set of all allocation sites in the program. Set $\mathcal{C} \subseteq S \cup \{\epsilon\}$ represents the space of all possible *object contexts*. For the rest of the paper we will use o_i or simply the index i to denote object context s_i. An instance method or constructor m is separately analyzed for context $o_i \in \mathcal{C}$, if m is invoked on an object allocated at site s_i. For simplicity, static methods (e.g., `main`) are analyzed in the special context ϵ.

Set $O' \subseteq S \cup S^2$ represents the set of replicated object names. If allocation site s_i appears in an instance method or constructor, there is a separate object name $o_{ij} \in O'$ for each context $o_j \in \mathcal{C}$ reaching the enclosing method of s_i. If s_i is enclosed by a static method, there is a single object name o_i (short for $o_{i\epsilon}$) that represents the objects created at that site. For the set of statements in Figure 3 all allocation sites occur in `main`; thus, set O' equals $\{o_1, o_2, o_3, o_4\}$. The separate analysis for reference variables is achieved by maintaining *context copies* of reference variables for each possible context. The set of replicated reference variables $R' \subseteq R \times \mathcal{C}$ is defined as follows. If r is a local variable in an instance method or constructor, r is mapped to a "fresh" variable r^c for every context $c \in \mathcal{C}$ that reaches m during the analysis. If $r \in R$ is a static variable or a local in a static method, r is mapped to itself. For the set of statements in Figure 3, constructors `B.B` and `A.A` are invoked on receiver o_3. Thus, context o_3 reaches `B.B` and `A.A`, and R' includes $\texttt{B.this}^{o_3}$, $\texttt{B.xb}^{o_3}$, $\texttt{A.this}^{o_3}$, and $\texttt{A.xa}^{o_3}$.

Effects of program statements. The analysis constructs a points-to graph with nodes from R' and O'. The effects of the program statements are essentially equivalent to analyzing each statement separately for each enclosing context. For example, if the analysis encounters direct assignment $l = r$ under context c, it infers that l^c may point-to the objects that r^c may point to. Similarly, if it encounters s_i: $l = new\ o_i$ under context o_j, it infers that o_{ij} appears in the points-to set of l^{o_j} (recall that the set of contexts coincides with the set of object creation sites).

At virtual call statements the analysis resolves the call separately for each context. When $l = r_0.m(r_1, ..., r_n)$ is encountered under context c, the analysis resolves the call based on each object o_{ij} in the points-to set of r_0^c. For each o_{ij} it determines an actual target method n based on the class of o_{ij} and the compile-time target m. The analysis then updates the set $\mathcal{C}_n$, which represents the set of reaching contexts of n (i.e., the set of contexts for which the statements in n are analyzed), with the new context o_i. It maps the formal parameters and return variable of n to the context of o_i and performs appropriate updates. For example, the analysis infers that formal parameter $p_j^{o_i}$ points to every object in the points-to set of r_j^c; similarly, l^c points to every object in the points-to set of return variable $ret_n^{o_i}$.

4.2 Framework Formulation

Object sensitivity as a dimension of precision can be expressed in the framework for annotated constraints by adding *object annotations* with appropriate operations.

Object annotations. The object annotations are tuples of the form $[i \times j]$ where i and j correspond (roughly) to object allocation sites. The index on the left describes the object context of the program element corresponding to the term on the left-hand side of a constraint; the one on the right describes the context of the element corresponding to the term on the right-hand side. Roughly, constraint $v_r \subseteq^{[i \times j]} v_l$ represents that the points-to set of r^{o_i} is a subset of the points-to set of l^{o_j}.

We use the wildcard symbol $*$ to efficiently represent all possible object contexts that may reach the term on the corresponding side of the constraint (i.e., context that may reach the enclosing method). It may appear on either side of an object annotation. There are five kinds of object annotations: (1) $[i \times j]$, (2) $[i \times *]$, (3) $[* \times j]$, (4) $[* \times *]$ and (5) $[\ \times\ ]$, where i and j are allocation site indices and $[\ \times\ ]$ is the empty object annotation, also denoted by ϵ_o. The meaning of the first four kinds is straightforward. For example, $v_r \subseteq^{[i \times j]} v_l$ represents that the points-to set of r^{o_i} is a subset of the points-to set of l^{o_j}. Similarly, $v_r \subseteq^{[i \times *]} v_l$ represents that for every possible context c reaching l, the points-to set of r^{o_i} is a subset of the points-to set of l^c.

The empty object annotation $[\ \times\ ]$ represents all possible contexts reaching the terms on the left and right-hand side of the constraint. In other words, constraint $v_r \subseteq^{[\ \times\ ]} v_l$ represents that for every context c of the enclosing method of r and l, the points-to set of r^c is a subset of the points-to set of l^c. The role of the empty object annotations is to model flow in contexts that may be unknown; this effectively avoids processing and generating constraints for program statements multiple times. For example, when the analysis processes statement 1 in Figure 3, the possible contexts of invocation of 1 are unknown and depend on the possible receivers of `A.A`. The analysis creates constraints $v_{A.this} \subseteq proj(ref, 3, u)$ and $v_{A.xa} \subseteq_f u$, where the empty annotation represents flow for all possible contexts of `A.A`; these constraints summarize intraprocedural flow and avoid processing of the statements in `A.A` multiple times.

Continuing with Figure 3, the semantics of the object-sensitive analysis specifies that the points-to set of `y` flows to the points-to set of `B.xb`^{o_3} and subsequently to `A.xa`^{o_3}. The formulation in our framework expresses this flow with constraints $v_y \subseteq^{[* \times 3]} v_{B.xb}$ and $v_y \subseteq^{[* \times 3]} v_{A.xa}$, where the only possible context of `y` is ϵ.

The operations on object annotations are as follows:
$match(s_1, s_2) \Rightarrow$

$$\begin{cases} \text{true} & \text{if } s_1 = \epsilon_o \text{ or } s_2 = \epsilon_o \\ \text{true} & \text{if } s_1 = [i \times j], s_2 = [k \times l] \text{ and } j = k \\ \text{true} & \text{if } s_1 = [i \times j], s_2 = [k \times l] \text{ and } j = * \text{ or } k = * \\ \text{false} & \text{otherwise} \end{cases}$$

Consider $v_x \subseteq^{s_1} v_y \subseteq^{s_2} v_z$. If s_1 is the empty object annotation, the constraints represent valid flow because the empty annotation can be instantiated to the context on the left side of s_2. If j and k are the same allocation site index, the two constraints can be combined to represent valid flow as well. Similarly, if j is $*$, the constraints represent valid flow, because the wildcard symbol matches every reaching context of y, including k. Clearly, if j and k are different allocation site indices the two constraints represent flow into and from two different context copies of y and cannot be combined to represent valid flow from x to z. If $match$ holds, the analysis performs concatenation:

$$concat(s_1, s_2) \Rightarrow \begin{cases} s_2 & \text{if } s_1 = \epsilon_o \\ s_1 & \text{if } s_2 = \epsilon_o \\ [i \times l] & \text{if } s_1 = [i \times j], s_2 = [k \times l] \end{cases}$$

Consider $v_x \subseteq v_y \subseteq^{[k \times j]} v_z$. The first constraint represents flow from x^c to y^c for every context c of the enclosing method of x and y; thus, it can be instantiated for k to represents flow from x^k to y^k. The new constraint $v_x \subseteq^{[k \times l]} v_z$ correctly represents flow from x^k to z^l. It remains to define $transpose$: $transpose([i \times j]) = [j \times i]$.

Constraint generation. For all statements, the analysis generates constraints with empty object annotations as shown in Figure 2. These constraints represent flow that is valid for all possible contexts of the method. During resolution, points-to sets and object annotations reaching the left-hand side of the constraint (e.g., v_r of $v_r \subseteq v_l$) are propagated to the right-hand side of the constraint (i.e., the flow is instantiated to the appropriate context). Consider Figure 3. Constraint $v_y \subseteq^{[* \times 3]} v_{A.xa}$ represents the fact that the points-to set of `y` in `main` flows to the points-to set of `A.xa`^{o_3}. Due to statement 1, a constraint $v_{A.xa} \subseteq u$ is generated as shown in Figure 2; it represents that for every enclosing context c of `A.A` the points-to set of `A.xa`^{c} flows to u^c. The empty annotation is instantiated and the resulting constraint represents correctly that the points-to set of `y` flows only to u^{o_3}.

In order to model the semantics of the object-sensitive $resolve$, the closure rule VIRTUAL from Section 3.1 is modified to handle object annotations. If object annotations s_1 and s_2 "match" in $ref(o_i, v_{o_i}, \overline{v_{o_i}}) \subseteq^{s_1} v$, and $v \subseteq_m^{s_2} lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$ the analysis consults the lookup table and based on the class of o_i and the compile-time target m finds a lambda term $lam(\overline{v_{p_0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_k}}, v_{ret})$ corresponding to the definition of the run-time target method. If $concat(s_1, s_2) = \epsilon_o$ the analysis creates constraints

$$ref(o_i, v_{o_i}, \overline{v_{o_i}}) \quad \subseteq^{[* \times i]} \quad v_{p_0}$$

$$lam(\overline{v_{p_0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_k}}, v_{ret}) \quad \subseteq^{[i\times *]} \quad lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$$

When $concat(s_1, s_2) = \epsilon_o$, allocation site s_i and the call site modeled by lambda term $lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l)$ appear in the same method. Therefore, for every enclosing context o_n of this method, object o_{in} flows to the points-to set of $p_0^{o_i}$. Similarly, the points-to set of each context copy $r_j^{o_n}$ flows to the points-to set of the corresponding formal $p_j^{o_i}$. This is reflected by constraints $ref(o_i, v_{o_i}, \overline{v_{o_i}}) \subseteq^{[*\times i]} v_{p_0}$ and $v_{r_j} \subseteq^{[*\times i]} v_{p_j}$.

In the case when $concat(s_1, s_2) = [X \times Y] \neq \epsilon_o$, object o_i flows to the call site represented by the lambda term due to some interprocedural flow. The analysis creates the following constraints:

$$\begin{array}{rcl} ref(o_i, v_{o_i}, \overline{v_{o_i}}) & \subseteq^{[X\times i]} & v_{p_0} \\ lam(\overline{v_{p_0}}, \overline{v_{p_1}}, \ldots, \overline{v_{p_k}}, v_{ret}) & \subseteq^{[i\times Y]} & lam(\overline{0}, \overline{v_{r_1}}, \ldots, \overline{v_{r_k}}, v_l) \end{array}$$

X represents object contexts $c \in \mathcal{C}$, the contexts of invocation of the enclosing method of s_i. Y represents the contexts of the method invocation site.

Example. This example illustrates object-sensitive analysis in our framework. Consider the statements in Figure 3. At line 6, the analysis creates constraints

$$ref(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq v_b \qquad v_b \subseteq_{B.B} lam(\overline{0}, \overline{v_y})$$

Applying the object-sensitive VIRTUAL results in:

$$\begin{array}{c} ref(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq^{[*\times 3]} v_{B.this} \\ lam(\overline{v_{B.this}}, \overline{v_{B.xb}}) \subseteq^{[3\times *]} lam(\overline{0}, \overline{v_y}) \end{array}$$

After applying the rules for non-atomic constraints in Figure 1 we have

$$ref(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq^{[*\times 3]} v_{B.this} \qquad v_y \subseteq^{[*\times 3]} v_{B.xb}$$

and after line 2 we have

$$ref(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq^{[*\times 3]} v_{B.this} \quad v_{B.this} \subseteq_{A.A} lam(\overline{0}, \overline{v_{B.xb}})$$

The analysis applies $concat$ on $s_1 = [* \times 3]$ and $s_2 = [\ \times\ ]$ and after appropriate context instantiation of $[\ \times\ ]$ it infers that object o_3 flows to the invocation site denoted by the lambda term only when the invocation occurs in the context of o_3. Applying the object-sensitive VIRTUAL followed by the rules in Figure 1 results in constraints

$$ref(o_3, v_{o_3}, \overline{v_{o_3}}) \subseteq^{[*\times 3]} v_{A.this}\ (1) \quad v_{B.xb} \subseteq^{[3\times 3]} v_{A.xa}\ (2)$$

Applying the analogous sequence of rules for o_4 at lines 7 and 3 results in

$$ref(o_4, v_{o_4}, \overline{v_{o_4}}) \subseteq^{[*\times 4]} v_{C.this} \qquad v_z \subseteq^{[*\times 4]} v_{C.xc}$$

and subsequently in

$$ref(o_4, v_{o_4}, \overline{v_{o_4}}) \subseteq^{[*\times 4]} v_{A.this}\ (3) \quad v_{C.xc} \subseteq^{[4\times 4]} v_{A.xa}\ (4)$$

At line 1 the analysis generates the following constraints:

$$v_{A.this} \subseteq proj(ref, 3, u)\ (5) \qquad v_{A.xa} \subseteq_f u\ (6)$$

Applying TRANS followed by the resolution rules for structural constraints to (1) and (5) results in $u \subseteq^{[3\times *]} v_{o_3}$. Similarly, from (3) and (5) we have $u \subseteq^{[4\times *]} v_{o_4}$. Clearly, the annotations are used to distinguish between flow from variable u in two separate object contexts. In the first case, the constraint represents flow from the context copy of u for object context o_3 and in the second case it represents flow from the context copy for o_4. Thus,

$$\begin{array}{c} v_y \subseteq^{[*\times 3]} v_{B.xb} \subseteq^{[3\times 3]} v_{A.xa} \subseteq_f u \subseteq^{[3\times *]} v_{o_3} \\ v_z \subseteq^{[*\times 4]} v_{C.xc} \subseteq^{[4\times 4]} v_{A.xa} \subseteq_f u \subseteq^{[4\times *]} v_{o_4} \end{array}$$

Clearly, $[4 \times 4]$ and $[3 \times *]$ do not "match" and the analysis avoids inferring constraints $v_{C.xc} \subseteq_f v_{o_3}$ and subsequently $ref(o_2, v_{o_2}, \overline{v_{o_2}}) \subseteq_f v_{o_3}$. The last constraint erroneously implies a field edge between o_3 and o_2. $[3 \times 3]$ and $[4 \times *]$ do not match and the constraint that implies a spurious field edge between o_4 and o_1 is avoided as well.

For simplicity we avoid discussion of handling of static variables, analysis complexity and analysis correctness. Detailed discussion of these issues occurs in [13].

4.3 Parameterized Object Sensitivity

For context-sensitive flow analyses, which are relatively expensive, it is important to propose parameterization mechanisms that enhance analysis flexibility. It should be possible to define analyses that achieve *targeted* context sensitivity by essentially selecting parts of the program for which keeping more precise information is likely to improve the quality of the flow information. Recent work has shown that targeted context sensitivity is crucial for flow-analysis-based production-strength software tools [17].

For the object-sensitive analysis we discuss two directions of parameterization; these and other directions are discussed in detail in [13]. First, the analysis designer can select the degree of precision in the context naming scheme. This is done by selecting a subset of the allocation sites to be used as contexts as described in Sections 4.1. For the rest of the sites, instead of the allocation site, the *class* of the allocated object is used as context. Clearly, this typically results in fewer contexts and more efficient analysis.

Second, the analysis can avoid analyzing certain statements separately for each enclosing context. For example, consider a virtual call $r_0.m(r_1, ..., r_n)$. The analysis may keep summary points-to sets r_0', r_1', ..., r_n', that contain the union of the points-to sets of r_0, r_1, ..., r_n for all possible contexts. When the call is processed, instead of examining the call in each enclosing context c as the semantics specifies (recall Section 4.1), the call is examined *once* and information is propagated once to each callee from the summary points-to sets. For example if o_{ij} appears in the points-to set of r_0' due to some r_0^c, instead of propagating r_j^c to $p_j^{o_i}$, the analysis propagates r_j' to $p_j^{o_i}$.

These parameterizations can be easily modeled in the framework for annotated constraints. For example, the second parameterization can be trivially modeled as follows: $lam(\overline{v_{p_0}}, \overline{v_{p_1}}, ..., \overline{v_{p_n}}, v_{ret}) \subseteq^{[i \times *]} lam(\overline{0}, \overline{v_{r_1}}, ..., \overline{v_{r_n}}, v_l)$; the wildcard annotations reflect that values are propagated to context copies $p_j^{o_i}$ from every possible context of the corresponding actual r_j.

5 Empirical Results

The goal of this experimental study is to address two questions. First, how costly is the addition of the annotations for modeling dimensions of precision? Second, what is the impact of the additional dimensions of precision on the flow analysis and its clients? The answers to these questions will provide insights about the usefulness of the framework as a basis for software tools.

Benchmarks. We performed experiments on a set of 23 publicly available Java programs. Column (1) in Table 1 shows the size of the programs, including library classes and methods, after using class hierarchy analysis (CHA) [5] to filter out irrelevant classes and methods[1]. The number of methods is essentially the number of nodes in the call graph computed by CHA. Details on the benchmarks and our analysis infrastructure appear in [13].

Framework instances. In order to answer the two questions stated above, we investigated four different points-to analyses and their formulations in the framework. The first one is a field-insensitive, flow- and context-insensitive points-to analysis referred to as *FieldInsens*; it is essentially a Java version of the analysis for C formulated using non-annotated inclusion constraints [6, 23]. The second one adds field sensitivity by using field annotations as described in Section 3; this analysis is referred to as *FieldSens*. Results from the detailed empirical investigation that contrasts the *FieldInsens* and *FieldSens* appear in [19, 13]. In summary, *FieldInsens* is on average about four times more expensive than *FieldSens*, and substantially less precise.

In addition, we investigated two object-sensitive analyses. The first one, denoted by *FullObjSens* adds object sensitivity to *FieldSens* by adding object annotations as described in Section 4. The fourth one, denoted by *ObjSens*, applies the following parameterizations: first, if there are more than fifty instances of a given class, the analysis uses the class name rather than the object name as context; second, for instance calls not through `this`, the analysis performs propagation using wildcard symbols as described at in Section 4.3. The detailed comparison between *FullObjSens* and *ObjSens* appears in [13]. In summary, *FullObjSens* is more expensive than *ObjSens* for some of the large programs, and exactly as precise. Table 1 presents a detailed comparison between *FieldSens* and *ObjSens*.

[1]CHA is an inexpensive analysis that determines the possible targets of a virtual call by examining the class hierarchy of the program.

Analysis cost. Column (2) of Table 1 shows the cost of *FieldSens* and *ObjSens*. All experiments were performed on a 900MHz Sun Fire-280R shared machine with 4Gb physical memory. For the majority of programs, the two analyses have comparable running times and memory usage. In certain cases, *ObjSens* actually performs better. This is consistent with previous results which show that *FieldSens* performs better than *FieldInsens*; in both cases the annotations seem to "cancel" edges that represents infeasible flow, which leads to smaller points-to sets and less work for the analysis. Thus, using annotations to model dimensions of precision can result in practical flow analysis.

Analysis precision. To address the second question we considered two client analyses of points-to analysis. *Modification side-effect analysis* determines, for each statement, the set of objects that may be modified by that statement. The analysis has a variety of uses in software tasks. It can be used to determine what methods are free of side-effects, and to compute coverage requirements for data-flow-based testing. For such applications side-effect analysis precision is essential because imprecise analysis typically results in wasted time for developers and testers. *Downcast safety analysis* determines, for each downcast statement, the set of objects that may be subject to the downcast; if all objects are of a class that is a subtype of the type in the cast, the downcast is determined to be safe. Since Java currently lacks generic types programmers have been using pseudo-generic classes expressed in terms of `Object` (e.g., Hashtable). Downcast safety analysis can be helpful in identifying errors due to incorrect use of pseudo-generic classes; also, it can be applied in tools for converting legacy Java source to use generics which will be added in Java 1.5.

Columns (3) in Table 1 summarizes the experiments with side-effect analysis. It shows the distribution of the number of modified objects per program statement for *FieldSens* and *ObjSens*. Each column corresponds to a specific range of numbers. For example, the first column corresponds to statements that may modify one, two or three objects, and the second column corresponds to statements that may modify at least 10 objects. Each column shows what percentage of the total number of statements that modify at least one object, corresponds to the particular range of numbers of modified objects. The measurements in Table 1 show that object sensitivity significantly improves analysis precision. For side-effect analysis based on *ObjSens*, on average 57% of the statements modify at most three objects. In contrast, for side-effect analysis based on *FieldSens* this percentage is 18%. It is significant to note that for *FieldSens* nearly 80% of the statements modify at least 10 objects. This is substantial imprecision that can be reduced by *ObjSens*.

Column (4) in Table 1 summarizes the experiments with

Program	(1) Program Size		(2) Analysis Cost				(3) Side-effect Analysis				(4) Downcast Safety	
	Classes	Methods	*FieldSens*		*ObjSens*		*FieldSens*		*ObjSens*		*FieldSens*	*ObjSens*
			Time	Mem	Time	Mem	1-3	$\geq$10	1-3	$\geq$10		
proxy	565	3283	4.8	35.1	5.3	34.8	19%	75%	76%	10%	24%	67%
compress	568	3316	8.3	39.6	10.1	40.1	23%	73%	68%	23%	24%	71%
db	565	3339	9.2	40.6	10.6	42.5	20%	76%	66%	25%	24%	74%
jb	574	3393	6.0	36.7	5.8	36.9	16%	80%	73%	12%	12%	44%
echo	577	3544	18.7	49.2	44.9	66.2	24%	69%	63%	26%	18%	43%
raytrace	582	3451	7.8	42.2	10.8	46.1	23%	72%	67%	24%	23%	71%
mtrt	582	3451	9.4	42.1	11.3	46.2	23%	72%	67%	24%	23%	71%
jtar	618	3583	16.8	50.3	24.4	58.9	19%	74%	62%	25%	17%	44%
jlex	578	3381	6.7	39.8	7.3	40.6	18%	79%	57%	10%	22%	78%
javacup	581	3564	23.2	55.8	21.2	58.5	14%	83%	54%	9%	9%	85%
rabbit	615	3770	9.1	46.2	11.7	45.6	20%	76%	48%	17%	23%	68%
jack	613	3573	28.7	54.8	24.9	56.7	17%	80%	54%	38%	15%	63%
jflex	608	3692	28.5	63.5	30.3	66.4	18%	78%	64%	13%	4%	62%
jess	715	3973	35.8	59.4	87.5	61.0	16%	79%	63%	29%	20%	73%
mpegaudio	608	3531	11.6	44.0	10.4	48.4	23%	78%	67%	24%	23%	68%
jjtree	620	4078	8.6	46.8	32.1	64.4	8%	90%	32%	42%	65%	80%
sablecc	864	5151	34.5	78.5	51.2	75.3	20%	77%	67%	20%	33%	47%
javac	730	4470	100.5	110.0	168.5	129.0	14%	83%	38%	42%	12%	36%
creature	626	3881	64.3	94.3	105.5	124.8	19%	79%	55%	32%	18%	33%
mindterm	686	4420	37.2	78.5	51.5	90.5	20%	73%	57%	30%	25%	47%
soot	1214	5669	139.4	117.8	115.9	117.9	31%	73%	46%	40%	17%	25%
muffin	894	5253	120.7	133.9	115.1	149.7	16%	80%	45%	49%	13%	35%
javacc	615	4198	99.6	96.6	93.4	101.9	10%	89%	29%	22%	6%	59%
Average							18%	78%	57%	36%	20%	58%

Table 1. Java programs and analysis results.

downcast safety analysis. *ObjSens* analyzes pseudo-generic classes such as containers context-sensitively and is able to resolve on average, almost 60% of the downcasts; in contrast *FieldSens* merges flow into containers and is able to resolve only 20% of the downcasts. Thus, object sensitivity is essential for downcast safety analysis.

These experiments indicate that adding dimensions of precision has substantial impact on the points-to analysis and its clients, while the analysis remains efficient and practical. Therefore, analyses based on annotated inclusion constraints can be usefully incorporated in software tools.

6 Related Work

In [19] we presented an ad-hoc formulation of field-sensitive analysis. In [14], we proposed object sensitivity and presented an implementation based on variable and constraint copying, not on annotated inclusion constraints. The implementation based on annotated constraints is clearly more general and flexible. For example, it allows precise object naming which is essential for the handling of containers and thus for downcast safety analysis. It is not implemented in [14]. The current paper introduces the framework for annotated constraints and uses the object-sensitive analysis only as an example analysis that can be expressed precisely and intuitively using annotated inclusion constraints. Our work is related to work in the following areas.

Constraint-based flow analysis. Fähndrich et al. [16, 7] present precise context-sensitive analysis for problems expressible with structural constraints. For non-structural constraints such as the inclusion constraints, the problem appears to be harder; context sensitivity cannot be expressed in the CFL reachability model from [16, 7].

Efficient implementations of Andersen's points-to analysis for C are presented in [6, 23] and [10]. These analyses are based on non-annotated constraints and do not model field or context sensitivity. Foster et al. present a context-sensitive points-to analysis for C based on Andersen's analysis using non-annotated inclusion constraints [8]. They conclude that (i) using an implementation based on constraint copying may result in impractical context-sensitive analysis and (ii) context sensitivity does not improve the precision of Andersen's analysis for C. Our experiments show that for object-oriented programs, field and context sensitivity are necessary to achieve useful points-to analyses because of inherent object-oriented features.

O'Callahan presents a unification-based context-sensitive alias analysis for Java [15] similar to [16, 7] that seems not to scale. Our analyses are inclusion-based and the experiments indicate that they may be more practical.

Points-to Analysis for Java. Recent work by Whaley and Lam [24] presents a context-sensitive points-to analysis using BDDs, based on the call-string approach to context sensitivity [21]. Our approach can model a variety of flow analyses and add dimensions of precision to them in the form of annotations; it can model both call-string-based and functional context sensitivity [21]). In addition, the approach from [24] appears to use a less precise object naming scheme than our analysis; as in our previous work [14], it distinguishes objects only by allocation sites which may be insufficient for downcast safety analysis.

In [20] Ruf presents flow-insensitive, context-sensitive alias analyses for Java. The analyses are based on the almost-linear unification-based Steensgaard's points-to analysis for C, and unlike our analyses, use bottom-up traversal on an approximate call graph, and method summaries to model context sensitivity. Other context-sensitive points-to analyses for Java are presented in [9]. In general, these analyses are more precise and significantly more costly than ours, due to their flow sensitivity. Flow-insensitive and context-insensitive points-to analyses for Java are described in [22, 12, 19, 11, 4].

7 Conclusions

We propose annotated inclusion constraints—a new general framework for formulating and implementing precise inclusion-based flow analyses. We implement several flow analyses as framework instances. The experiments show that adding precision dimensions as annotations has substantial impact on analysis precision, while the analysis remains practical. We plan to investigate analyses expressible in the framework and their impact on software tasks.

References

[1] A. Aiken, M. Fähndrich, J. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Workshop on Types in Compilation*, pages 78–96, 1998.

[2] A. Aiken and E. Wimmers. Solving systems of set constraints. In *LICS*, pages 329–340, 1992.

[3] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *FPCA*, pages 31–41, 1993.

[4] M. Berndl, O. Lhotak, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDD's. In *PLDI*, pages 103–114, 2003.

[5] J. Dean, D. Grove, and C. Chambers. Optimizations of object-oriented programs using static class hierarchy analysis. In *ECOOP*, pages 77–101, 1995.

[6] M. Fähndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI*, pages 85–96, 1998.

[7] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI*, pages 253–263, 2000.

[8] J. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *SAS*, LNCS 1824, pages 175–198, 2000.

[9] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *OOPSLA*, pages 108–124, 1997.

[10] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *PLDI*, pages 254–264, 2001.

[11] O. Lhotak and L. Hendren. Scaling Java points-to analysis using Spark. In *CC*, pages 153–169, 2003.

[12] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *PASTE*, pages 73–79, 2001.

[13] A. Milanova. *Precise and Practical Flow Analysis of Object-Oriented Software*. PhD thesis, Rutgers University, Aug. 2003.

[14] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM TOSEM*, 14(1):1–42, 2005.

[15] R. O'Callahan. *The Generalized Aliasing as a Basis for Software Tools*. PhD thesis, Carnegie Mellon University, 2000.

[16] J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *POPL*, pages 54–66, 2001.

[17] D. Reimer, E. Schonberg, K. Srinivas, H. Srinivasan, B. Alpern, R. D. Johnson, A. Kershenbaum, and L. Koved. SABER: smart analysis based error reduction. In *ISSTA*, pages 243–251, 2004.

[18] A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *PLDI*, pages 47–56, 2000.

[19] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *OOPSLA*, pages 43–55, 2001.

[20] E. Ruf. Effective synchronization removal for Java. In *PLDI*, pages 208–218, 2000.

[21] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[22] M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an emprirical comparison. Technical report, U. Passau, Sept. 2000.

[23] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *POPL*, pages 81–95, 2000.

[24] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, 2004.